Optimizing Performance with GraalVM

Alina Yurenko
GraalVM Developer Advocate
Oracle Labs
November 11, 2019
Safe harbor statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle’s products may change and remains at the sole discretion of Oracle Corporation.

GraalVM Native Image Early Adopter Status

GraalVM Native Image technology (including SubstrateVM) is early adopter technology. It is available only under an early adopter license and remains subject to potentially significant further changes, compatibility testing and certification.
Agenda

1. Performance metrics
2. JIT compilation
3. AOT compilation
4. Demo: JIT and AOT
5. Tools
Performance metrics
How to measure performance?

- Throughput
- Latency
- Capacity
- Utilization
- Efficiency
- Scalability
- Degradation

https://learning.oreilly.com/library/view/optimizing-java -
by Chris Newland, James Gough, Benjamin J Evans
Performance metrics

• Throughput
• Efficiency
• Scalability
Optimizing performance with GraalVM
Universal Virtual Machine

1. Run programs more efficient
2. Make developers more productive
GraalVM Compiler

- Brand new compiler written itself in Java;
- Adds new optimizations on top of traditional ones;
- Supports multiple languages and platforms;
- Can work in JIT & AOT modes.
GraalVM Project Goals

1. High performance for abstractions of any language
2. Low-footprint ahead-of-time mode for JVM-based languages
3. Convenient language interoperability and polyglot tooling
4. Simple embeddability in native and managed programs
GraalVM

JIT

java MyMainClass

AOT

native-image MyMainClass
./mymainclass
Java Dynamic Execution
How GraalVM AOT works

Input:
- All application classes, libraries, and VM
- Application
- Libraries
- JDK
- Substrate VM

Iterative analysis until fixed point is reached

Points-to Analysis
Run Initializations
Heap Snapshotting

Ahead-of-Time Compilation
Code in Text Section
Image Heap in Data Section
Image Heap Writing

Output:
A native executable

Legend:
-
-
AOT vs JIT: Startup Time

JIT
- Load JVM executable
- Load classes from file system
- Verify bytecodes
- Start interpreting
- Run static initializers
- First tier compilation (C1)
- Gather profiling feedback
- Second tier compilation (GraalVM or C2)
- Finally run with best machine code

AOT
- Load executable with prepared heap
- Immediately start with best machine code
AOT vs JIT: Memory Footprint

JIT
- Loaded JVM executable
- Application data
- Loaded bytecodes
- Reflection meta-data
- Code cache
- Profiling data
- JIT compiler data structures

AOT
- Loaded application executable
- Application data
Demo: startup and memory footprint
AOT vs JIT: Peak Throughput

JIT
- Profiling at startup enables better optimizations
- Can make optimistic assumptions about the profile and deoptimize

AOT
- Needs to handle all cases in machine code
- Profile-guided optimizations help
- Predictable performance
Demo: peak performance
How to achieve even more with native images: PGO

The GraalVM compiler is built ground-up with profiles in mind.
Collecting profiles is essential for performance of native images.
Profile guided optimizations requires running relevant workloads before building an image.

```bash
$ java -Dgraal.PGOInstrument=myclass.iprof MyClass
$ native-image --pgo=myclass.iprof MyClass
$ ./myclass
```
Native Image: Profile-Guided Optimizations (PGO)

Handled requests per second

- Native Image (EE with PGO)
- Native Image (CE)
- JDK 8, Java HotSpot VM

Cumulative number of requests sent by ApacheBench
AOT vs JIT: Max Latency

**JIT**
- Many low latency GC options available
- G1
- CMS
- ZGC
- Shenandoah

**AOT**
- Only regular stop&copy collector
- Assumes small heap configuration
- Can quickly restart; could use load balancer instead of GC
AOT

Startup Speed
Peak Throughput
Low Memory Footprint
Reduced Max Latency
Small Packaging

JIT
GraalVM JIT Performance: Renaissance.dev

Speedup vs JDK8

- akka-uct
- db-shootout
- finagle-chirper
- future-genetic
- movie-lens
- page-rank
- scrabble
- geomean

EE/C2
CE/C2
Scala Performance

https://medium.com/graalvm/compiling-scala-faster-with-graalvm-86c5c0857fa3
Microservice Frameworks: Startup Time

- **Helidon**: 35 ms
  - Native Image: 1030 ms
  - JDK 12: 988 ms
  - JDK 8: 952 ms
- **Micronaut**: 37 ms
  - Native Image: 2087 ms
  - JDK 12: 2101 ms
- **Quarkus**: 16 ms
  - Native Image: 952 ms
  - JDK 12: 940 ms
  - JDK 8: 952 ms
Microservice Frameworks: Memory Usage

<table>
<thead>
<tr>
<th>Framework</th>
<th>Native Image</th>
<th>JDK 12</th>
<th>JDK 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helidon</td>
<td>31 MB</td>
<td>116 MB</td>
<td>106 MB</td>
</tr>
<tr>
<td>Micronaut</td>
<td>41 MB</td>
<td>172 MB</td>
<td>180 MB</td>
</tr>
<tr>
<td>Quarkus</td>
<td>17 MB</td>
<td>125 MB</td>
<td>121 MB</td>
</tr>
</tbody>
</table>
Spring Boot Applications as GraalVM Native Images

https://www.youtube.com/watch?v=3eoAxphAU1g
Spring Boot Applications as GraalVM Native Images

“Spring Graal Native” project: https://github.com/spring-projects-experimental/spring-graal-native

```
Alinas-MacBook-Pro:~/spring-graal-native/spring-graal-native-samples$ ls
commandlinerunner  spring-petclinic-jpa  vanilla-orm2
commandlinerunner-maven springmvc-tomcat  vanilla-rabbit
kotlin-webmvc  vanilla-grpc  vanilla-thymeleaf
logger  vanilla-jpa  vanilla-tx
messages  vanilla-orm  webflux-netty
```
Demo: native Spring Boot app
Currently

AOT

- Startup Speed
- Peak Throughput
- Reduced Max Latency
- Small Packaging
- Low Memory Footprint
AOT

- Startup Speed
- Peak Throughput
- Low Memory Footprint
- Reduced Max Latency
- Small Packaging

Goal
Simplifying the Native Image Configuration

```
mkdir -p META-INF/native-image

$JAVA_HOME/bin/java -agentlib:native-image-agent=config-output-dir=META-INF/native-image HelloReflection foo xyz

[
  {
    "name":"HelloReflection",
    "methods":[{ "name":"foo", "parameterTypes":[] }]
  }
]
```

Continue Learning About GraalVM Native Images

- GraalVM Native Images: The Best Startup Solution for Your Applications: [https://www.youtube.com/watch?v=z0jedLjcWjl](https://www.youtube.com/watch?v=z0jedLjcWjl)
GraalVM Performance Workshop: Thursday, 1 PM - 4 PM

Maximizing Performance with GraalVM

Christian Wimmer
Project Lead of the Native Image Part of GraalVM @graalvm
Summary

GraalVM JIT
- Peak throughput
- Max Latency
- No configuration

GraalVM AOT
- Startup Time
- Memory footprint
- Packaging size
Tools
Ideal Graph Visualizer

```java
443 // Generate two digits per iteration
444 int q = 1 / 100;
445 // Finally: i = i + (q <= 6) + (q <= 5) + (q <= 2));
446 r = 1;
447 i = q;
448 buf[---CharPos] = Digits[i];
449 buf[---CharPos] = Digits[i];
450 }
451 // Fail thru to test mode for smaller numbers
452 // assert(i <= 0x500, s);
453 for (;i;)
454 }
```
Do even more with GraalVM
JavaScript + Java + R
Demo: Java + JavaScript
Do even more with GraalVM: Cross-Platform Development

IDE

Plugin

JavaFX mobile

Java static libs

Gluon Mobile

GluonTools

Graal VM Native Image
Industry Use Cases
Twitter uses GraalVM compiler in production to run their Scala microservices.
• Peak performance: +10%
• Garbage collection time: -25%
• Seamless migration

Oracle®
Cloud Infrastructure
The rich ecosystem of CUDA-X libraries is now available for GraalVM applications.

GPU kernels can be directly launched from GraalVM languages such as R, JavaScript, Scala and other JVM-based languages.
What’s next for GraalVM
What’s next for GraalVM

• JDK-11 based builds;
• ARM64 and Windows support;
• Low-latency, high-throughput, and parallel GC for native images;
• Work with the community to support important libraries;
• New languages and platforms;
• Your choice – contribute!
Get Started

- Downloads
- Documentation
- Community support
GraalVM Versions

Community Edition

GraalVM Community is available for free for evaluation, development and production use. It is built from the GraalVM sources available on GitHub. We provide pre-built binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is experimental.

Download from GitHub

Enterprise Edition

GraalVM Enterprise provides additional performance, security, and scalability relevant for running applications in production. It is free for evaluation uses and available for download from the Oracle Technology Network. We provide binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is experimental.

Download from OTN
What’s next for you

- Download: graalvm.org/downloads
- Follow updates: @GraalVM / #GraalVM
- If you need help:
  - graalvm.org/community
  - graalvm-users @oss.oracle.com
Key Takeaways

• Write small methods;
• Local allocations are free, global data structures expensive;
• Don’t hand optimize, unless you have studied the compiler graph;
• For best throughput use **GraalVM JIT**,
• for best startup & footprint use **GraalVM AOT**.
Thank you!

Alina Yurenko / @alina_yurenko
GraalVM
Oracle Labs